Application Report
SPRAAU8A–March 2008–Revised August 2017

Copying Compiler Sections From Flash to RAM on the TMS320F28xxx DSCs

Tim Love and Vamsikrishna Gudivada

ABSTRACT

This application report and associated code files provide functionality for copying initialized compiler sections from the internal Flash memory to the internal random access memory (RAM) of the TMS320F28xxx digital signal controllers (DSCs) at run time for optimizing execution speed. The solution provided implements this functionality directly after booting before entering the c_int00 C entry routine.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/spraau8.

Contents

List of Figures

List of Tables

Trademarks

1 Introduction

In many applications, code execution speed is critical to the end application. A few examples of time critical end equipment would be medical, motion control, motor control, etc. Many of these applications use the TMS320F28xxx DSCs due to its internal Flash memory. The internal Flash memory is a great benefit of the TMS320F28xxx family because it is non volatile memory that allows designers to store application code internal to the chip as opposed to interfacing external memory to store this code. The downside of using the internal Flash is that wait states are required to access the flash, which leads to slower code execution time. In most applications this is not an issue. Other applications may require zero wait-states for maximum speed. The internal RAM memory has zero wait-states but is a volatile memory. As a result, initialized sections cannot be stored on this memory for boot up.

The solution presented allows designers to copy initialized compiler sections (.text, .cinit, .econst, .switch, etc.) from the Flash to RAM at runtime to allow maximum execution speed. This gives code execution a boost from up to 15 wait-states to 0 wait-states. For another solution on copying just certain functions from Flash to RAM, see *Running an Application from Internal Flash Memory on the TMS320F28xx DSP* (SPRA958). This implementation should be used in most C2000™ DSC applications. Other applications requiring tight timing with continuous zero wait-states should implement this presented solution.

An assembly routine was written to perform the copy from Flash to RAM. This assembly code is executed after the reset vector before the call to c_int00. This ensures that the sections are copied before c_int00 calls main().

Some projects are small enough that all of the initialized sections can be copied to RAM. Other projects, however, have initialized sections that are larger than the max 18K of internal RAM for TMS320F281x/TMS320F280xx DSCs and 34K for TMS320F2833x DSCs. These projects may not be able to copy all initialized sections to RAM but could use this solution to copy some of the sections.

It is assumed that *Running an Application from Internal Flash Memory on the TMS320F28xx DSP* (SPRA958) has been viewed and its methodologies are followed for Flash implementation.

2 Compiler Sections

The compiler creates multiple portions of code and data called sections. These sections are categorized into two different groups: initialized and uninitialized. The initialized group of sections is composed of all code, constants, and initialization tables. Table 1 shows the initialized sections produced by the compiler.

Table 1. Initialized Sections

Name	Contents	Restrictions
.cinit	Tables for explicitly initialized global and static variables	Program
.const	Global and static const variables that are explicitly initialized and string literals	Low 64K data
.econst	Far constant variables	Anywhere in data
.pinit	Tables for global object constructors	Program
.switch	Tables for implementing switch statements	Program (with -mt option) Data (without -mt option)
.text	Executable code and constants	Program

The uninitialized group of sections is composed of variables, the stack, and malloc memory. Table 2 shows the uninitialized sections produced by the compiler.

Table 2. Uninitialized Sections

Name	Contents	Restrictions
.bss	Global and static variables	Low 64K data
.ebss	Far global/static variables	Anywhere in data
.stack	Stack space	Low 64K data
.sysmem	Memory for malloc functions	Low 64K data
.esysmem	Memory for far_malloc functions	Anywhere in data

Once the compiler has generated these sections the linker takes the individual sections from each source file and combines them to create an output section. The linker command file (.cmd) is used to tell the linker where to allocate these sections. Initialized sections must be assigned to a non volatile memory like Flash/ROM so the application is not erased when power is removed from the target. Uninitialized sections can be allocated to RAM as they are initialized during code execution.

For more information regarding compiler sections and linking, see the *TMS320C28x Assembly Language Tools User's Guide* (SPRU513) and the *TMS320C28x Optimizing C/C++ Compiler User's Guide* (SPRU514).

Texas Instruments has multiple examples available that show the use of the linker command file to allocate compiler sections. One such example is the *Running an Application from Internal Flash Memory on the TMS320F28xx DSP* (SPRA958). This application report provides examples that demonstrate the use of the linker command file for both RAM based and Flash based projects.

The C/C++ Header Files and Peripheral Examples for each C28x™ DSP generation also provides examples for RAM and Flash based examples. For more information, see *C281x C/C++ Header Files and Peripheral Examples* (SPRC097), *C280x, C2801x C/C++ Header Files and Peripheral Examples Software Tools* (SPRC191), and C2833x/C2823x C/C++ Header Files and Peripheral Examples Application Software (SPRC530).

3 Software

The associated code files for this application report includes a modified version of the CodeStartBranch.asm file provided with the C/C++ Header Files and Peripheral Examples, as well as the DSP28xxx_SectionCopy_nonBIOS.asm file used for copying sections in a non DSP/BIOS™ based project. The ready made linker command files for each TMS320F28xxx DSC generation are also provided. Example projects are supplied as well to demonstrate the use of these files. The TMS320F2808 is referenced for the software portion of this application report.

The software is self contained and will extract with the F28xxx_Flash_to_Ram folder as the base directory. This code uses several files from the C/C++ Header Files and Peripheral Examples and was tested with Code Composer Studio™ software version 3.3 using F28xxx Code Generation Tools version 5.0.0B3.

3.1 Description

The general software flow for this functionality is: code_start → wd_disable → copy_sections → c_int00 → main(). This software flow only differs from a standard application software flow by calling the copy_sections routine. The standard flow is code_start → wd_disable → c_int00 → main().

3.1.1 Code_start and wd_disable

The code_start and wd_disable routines are provided in the DSP28xxx_CodeStartBranch.asm file. After power up, the code_start routine executes since it is allocated to the Flash boot address of 0x3F7FF6. For more information, see *Running an Application from Internal Flash Memory on the TMS320F28xx DSP* (SPRA958). This routine is shown below:

```
WD_DISABLE    .set 1          ;set to 1 to disable WD, else set to 0
    .ref copy_sections
    .global code_start
;***********************************************************************
* Function: codestart section
*
* Description: Branch to code starting point
;***********************************************************************
    .sect "codestart"

code_start:
    .if WD_DISABLE == 1
        LB wd_disable         ;Branch to watchdog disable code
    .else
        LB copy_sections      ;Branch to copy_sections
    .endif
```

This function was modified from the original CodeStartBranch.asm file provided with the C/C++ Header Files and Peripheral Examples by only changing the second call to copy_sections instead of _c_int00. This call will only be made if the WD_DISABLE is 0. As shown above, the code sets WD_DISABLE to 1. This causes a branch to the wd_disable routine. This routine is shown below:

```
;*****************************************************************
* Function: wd_disable
*
* Description: Disables the watchdog timer
;*****************************************************************
     .if WD_DISABLE == 1

     .sect "wddisable"
wd_disable:
     SETC OBJMODE            ;Set OBJMODE for 28x object code
     EALLOW                  ;Enable EALLOW protected register access
     MOVZ DP, #7029h>>6      ;Set data page for WDCR register
     MOV @7029h, #0068h      ;Set WDDIS bit in WDCR to disable WD
     EDIS                    ;Disable EALLOW protected register access
     LB copy_sections        ;Branch to copy_sections

     .endif
```

This is required as the watchdog should be disabled during the copy_sections and c_int00 function execution, otherwise the watchdog could timeout before main( ) is entered. This function was also modified from the original CodeStartBranch.asm file provided with the C/C++ Header Files and Peripheral Examples. The only modification is a branch to copy_sections instead of the c_int00 routine.

3.1.2 Copy_sections

The copy_sections routine is provided in the DSP28xxx_SectionCopy_nonBIOS.asm file. Once execution is to this phase, the watchdog has been disabled and the sections are ready to be copied. The copy for each section is prepared by storing the size of the section into the accumulator followed by storing the load address and run address into the XAR6 and XAR7 registers, respectively. An example of this functionality is as follows:

```
MOVL XAR5,#_text_size          ; Store Section Size in XAR5
MOVL ACC,@XAR5                 ; Move Section Size to ACC
MOVL XAR6,#_text_loadstart     ; Store Load Starting Address in XAR6
MOVL XAR7,#_text_runstart      ; Store Run Address in XAR7
LCR copy                       ; Branch to Copy
```

NOTE: The size, loadstart, and runstart symbols are all generated by the linker. This is discussed in the Memory Allocation – Linker Command Files section.

After the addresses and size has been stored, the copy subroutine is called to determine if the section was created by the compiler. This is tested by determining if the accumulator is 0:

```
copy:
     B return,EQ               ; Return if ACC is Zero (No section to copy)
     SUBB ACC,#1

     RPT AL                    ; Copy Section From Load Address to
     || PWRITE *XAR7, *XAR6++  ; Run Address

return:
     LRETR                        ; Return
```

If the accumulator is 0, execution is returned to the calling address. If the accumulator is not zero, the section needs to be copied. This is performed by the PWRITE instruction as shown above. The PWRITE copies the memory pointed to by XAR6 to XAR7. In this case, from the load address to the run address. This continues until the accumulator is zero indicating the end of the section. Once all sections have been copied, a branch to the c_int00 routine is performed as shown.

```
LB _c_int00 ; Branch to start of boot.asm in RTS library
```

At this point, the C environment is setup and main() is entered. For complete copy_sections routine listing, see the DSP28xxx_SectionCopy_nonBIOS.asm file in the associated code files.

3.1.3 Memory Allocation – Linker Command Files

As discussed in Section 2, the linker command file tells the linker where to allocate the compiler generated sections. The C/C++ Header Files and Peripheral Examples provide standard linker command files for use in applications.

Three linker command files are supplied in the associated code files to configure the memory allocation:

- F280xx_nonBIOS_flash.cmd
- F281x_nonBIOS_flash.cmd
- F2833x_nonBIOS_flash.cmd

Each file is generally written in the same manner with small differences in the memory layout (device specific). The MEMORY portion of the linker command file defines the memory available on the device to linked sections. The memory map of the device is used for this process. The memory map is found in the device-specific datasheet. For more information, see *TMS320F2810, TMS320F2811, TMS320F2812, TMS320C2810, TMS320C2811, TMS320C2812 Digital Signal Processors Data Manual* (SPRS174O), *TMS320F2809, F2808, F2806, F2802, F2801, C2802, C2801, and F2801x DSPs Data Manual* (SPRS230), and *TMS320F28335, TMS320F28334, TMS320F28332 TMS320F28235, TMS320F28234, TMS320F28232 Digital Signal Controllers (DSCs) Data Manual* (SPRS439).

Figure 1 shows the memory map of the TMS320F2808.

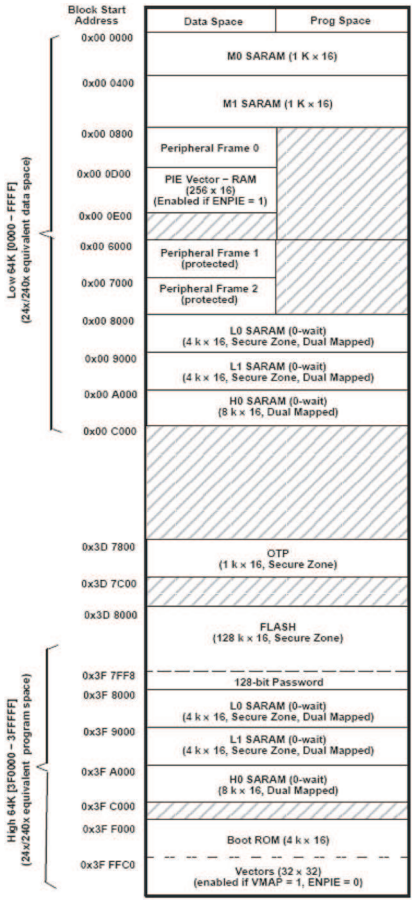

Figure 1. TMS320F2808 Memory Map

The TMS320F28xxx DSCs contain RAM internally to the device that can be allocated in single sections or in larger expanded sections, since it is predominately contiguous within the memory map. As shown in the memory map, the F2808 contains L0, L1, and H0 SARAMs mapped in contiguous memory spaces allowing the creation of one large block of memory. This RAM block can be defined within the MEMORY portion of the .cmd file as follows:

```
RAM_H0L0L1 : origin = 0x008000, length = 0x004000 /* on-chip RAM */
```

The rest of the memory would also be defined in the MEMORY section. For an example of complete memory allocation, see the linker command files included in the associated code files.

The second portion of the linker command file is the SECTIONS specification. This is where the actual compiler sections are linked to the memory areas. All of the sections from the DSP28xxx_CodeStartBranch.asm and DSP28xxx_SectionCopy_nonBIOS.asm are loaded and ran from the Flash memory. This allocation is shown below.

```
codestart        : > BEGIN_FLASH, PAGE = 0   /* Used by file CodeStartBranch.asm */
wddisable        : > FLASH_AB, PAGE = 0      /* Used by file CodeStartBranch.asm */
copysections     : > FLASH_AB, PAGE = 0      /* Used by file SectionCopy.asm */
```

The other initialized compiler sections are allocated to load to the Flash but run from the internal RAM. This is achieved by specifying the LOAD and RUN directives. An example of this allocation is shown below.

```
.text : LOAD = FLASH_AB, PAGE = 0            /* Load section to Flash */
        RUN = RAM_H0L0L1,PAGE = 0            /* Run section from RAM */
        LOAD_START(_text_loadstart),
        RUN_START(_text_runstart),
        SIZE(_text_size)
```

To gain access to the specific addresses associated with a section, the LOAD_START, RUN_START, and SIZE address and dimension directives are used as shown above. The addresses and size produced by these directives are used by the DSP28xxx_SectionCopy_nonBIOS.asm file to point to the correct addresses during the copy. DSP28xxx_SectionCopy_nonBIOS.asm references these values by creating global variables as shown below.

```
.global _cinit_loadstart, _cinit_runstart, _cinit_size
.global _const_loadstart, _const_runstart, _const_size
.global _econst_loadstart, _econst_runstart, _econst_size
.global _pinit_loadstart, _pinit_runstart, _pinit_size
.global _switch_loadstart, _switch_runstart, _switch_size
.global _text_loadstart, _text_runstart, _text_size
```

For more information regarding linker command files and the address and dimension operators, see the *TMS320C28x Assembly Language Tools User's Guide* (SPRU513).

3.2 Testing Example

The examples provided were tested with the TMS320F2812, TMS320F2808, and TMS320F28335 eZdsp development boards. The led will blink on the example project boards for visual confirmation that the code is working appropriately. The following procedures were used to program and test the project with the F2808 eZdsp. The same procedure can be used for the other eZdsp development boards as well.

3.2.1 Code Composer Studio Environment

1. Connect the F2808 eZdsp to the PC using the on board USB connection and power the board with the supplied power connector.
2. Start Code Composer Studio with the F2808 eZdsp emulation driver selected in the CCS setup utility.
3. Open and Build the Example_280xx_Flash_to_RAM_nonBIOS.pjt by selecting Project → Open followed by Project → Rebuild All.
4. Program the resulting .out file to the Flash by using the CCS On Chip Flash Programmer from the Tools menu. If this is currently not installed, it can be downloaded from the Update Advisor.
5. Load Symbols to debug the program by selecting File → Load Symbols → Load Symbols Only.
6. Run Program by selecting Debug → Run.

The LED on the eZdsp should be flashing to indicate that the program is running.

NOTE: If breakpoints are required to halt the operation during code execution, they should not be set until the copy_section routine has executed. If the breakpoints are set before, the copy_section routine will copy code over the set breakpoint and execution will not halt.

3.2.2 Standalone Operation

1. Follow the procedures given in Section 3.2.1 to program the example to the eZdsp development board.
2. Close Code Composer Studio and disconnect the USB cable.
3. Configure SW1 for Boot to Flash mode. For more information, see the *eZdsp™ F2808 USB Technical Reference* (www.spectrumdigital.com).
4. Remove power and reapply power to the eZdsp.

The LED on the eZdsp should be flashing to indicate that the program is running.

3.3 Application Integration

An existing Flash application can easily be migrated for this functionality with the associated code files. The basic migration procedure is as follows:

1. Replace the existing CodeStartBranch.asm source file with the supplied DSP28xxx_CodeStartBranch.asm.
2. Add the DSP28xxx_SectionCopy_nonBIOS.asm to the project.
3. Replace the current linker command file with the generation-specific supplied linker command file.

This basic migration procedure does not take into account application-specific situations such as user-defined sections, other section allocations, etc.

3.3.1 Example Integration

To demonstrate the application integration process, the Example_2808_Flash.pjt from the C280x, C2801x C/C++ Header Files and Peripheral Examples can be migrated using the following procedure.

1. Download and install the C280x, C2801x C/C++ Header Files and Peripheral Examples. For more information, see *C280x, C2801x C/C++ Header Files and Peripheral Examples Software Tools* (SPRC191).
2. Connect the board and open the project as described in Steps 1-3 of Section 3.2.1.
3. Remove the DSP280x_CodeStartBranch.asm file by right clicking on the file and selecting Remove from Project. Replace it with the DSP28xxx_CodeStartBranch.asm by selecting Project → Add Files to Project.
4. Add the DSP28xxx_SectionCopy_nonBIOS.asm to the project by selecting Project → Add Files to Project.
5. Remove the F2808.cmd file by right clicking on the file and select Remove from Project. Replace it with the F280xx_nonBIOS_flash.cmd by selecting Project → Add Files to Project.
6. Change .sect “ramfuncs” located in the DSP280x_usDelay.asm to .text to allocate the DSP28x_usDelay routine to the .text section.
7. Remove the #pragma CODE_SECTION(InitFlash, “ramfuncs”); code from the DSP280x_SysCtrl.c file. This will allocate the InitFlash( ) function to the .text section instead of ramfuncs.
8. Remove the #pragma CODE_SECTION(epwm1_timer_isr, “ramfuncs”); and #pragma CODE_SECTION(epwm2_timer_isr, “ramfuncs”); source lines from the Example_280xFlash.c. This will allocate the two ISRs to the .text section instead of ramfuncs.
9. Remove the MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);, and InitFlash( ); source lines from the Example_280xFlash.c file. These are not required as the code will already be copied to RAM.
10. Program and run the project as described in steps 4-6 of Section 3.2.1.

The LED on the eZdsp should be flashing to indicate that the program is running. The standalone operation can also be tested following the steps in Section 3.2.2.

NOTE: This project contains a user-defined section named ramfuncs. This is not needed since it is only used to copy code sections once in the main( ) function. The ramfuncs allocations are located in the DSP280x_usDelay.asm, DSP280x_SysCtrl.c, and Example_280xFlash.c files.

4 Benchmarks, Limitations, and Suggestions

Different applications have different requirements whether it be execution time, memory capacity, ease of use, etc. The solution presented adds to some requirements but may not be able to meet other requirements; therefore, memory usage, execution time, integration, etc., should all be considered when implementing this functionality.

4.1 Memory Usage

The only section that adds to the memory usage is the copy_sections routine from the DSP28xxx_SectionCopy_nonBIOS.asm file. As supplied, this file only adds 0x3C of allocated memory within the internal Flash. The code_start and wd_disable functions will not add extra memory allocation as these sections are used with all C2000 projects as shown in the C/C++ Header Files and Peripheral Examples.

4.2 Benchmarks

Since this functionality is implemented directly after boot, the Flash wait states and phase-locked loop (PLL) are not configured and, therefore, run at their default values. The Flash wait states are configured for 15 cycles and SYSCLKOUT of OSCCLK/2 for the F280xx/F281x devices and OSCCLK/4 for F2833x devices. Using the profiling feature of Code Composer Studio, the execution time can be measured. Table 3 shows the timing information for each F28xxx DSC by measuring the time elapsed from boot to the first instruction within the main( ) function for the examples provided in the associated code files. As shown, the execution time for each platform increases; this is due to an increase in code size and decreasing SYSCLKOUT.

Table 3. Execution Time

	OSCCLK (MHz)	SYSCLKOUT (MHz)	Cycles	Execution Time (ms)
TMS320F2812	30	15	18,576	1.238
TMS320F2808	20	10	20,560	2.056
TMS320F28335	30	7.5	29,681	3.957

4.3 Limitations

The limiting factor for this implementation is the amount of internal RAM available on the TMS320F28xxx DSCs. This limits what projects can implement this functionality. If the project is too large to fit in the RAM, this functionality cannot be implemented.

Since the C2000 platform is geared towards motor control and digital power applications, Texas Instruments provides numerous software packages for these types of applications (References 11,12,13). For more information, see F280x Motor-Specific Software Solutions - APSF280x (http://focus.ti.com/docs/toolsw/folders/print/apsf280x.html), TMS320F281x Motor-Specific Software Solutions (http://focus.ti.com/dsp/docs/dspplatformscontento.tsp?sectionId=2&familyId=1406&tabId=2027), and the TMS320C2000™ Controller Digital Power Software Library (http://focus.ti.com/dsp/docs/dspcontent.tsp?contentId=25262).

As supplied, these software packages run from the internal RAM. This demonstrates that these control systems can be run from the internal RAM as the base project is small enough to be allocated to this memory.

4.4 Suggestions

- In applications requiring this functionality, not all initialized compiler sections may need to be copied to RAM or have enough RAM to copy all the sections. The application code itself may only need to be copied. In this case, only the .text section would be copied to the RAM. For this, all of the copy routines for the other sections can be removed from the DSP28xxx_SectionCopy_nonBIOS.asm file and the linker command file can be changed to load and run the other sections from Flash. This will save Flash space and cut down on the execution time required before main( ) is entered.
- Determine that the application can handle the small lag in execution time to copy the sections. If the application cannot handle this time, the procedure shown in *Running an Application from Internal Flash Memory on the TMS320F28xx DSP* (SPRA958) should be used to copy just certain portions of the code to RAM.
- If using DSP/BIOS, the methodologies from *Running an Application from Internal Flash Memory on the TMS320F28xx DSP* (SPRA958) for copying certain portions of code to RAM is suggested. A project that uses DSP/BIOS is generally a larger application and this solution is not recommended.

5 Conclusion

This application report has proven that the TMS320F28xxx DSCs can achieve zero wait state execution by copying compiler sections from internal Flash memory to internal RAM at runtime before C entry. This solution also shows that this is directly limited by code size and memory size. The associated code files provide designers with a ready-made solution to implement this functionality within their design.

6 References

1. *Running an Application from Internal Flash Memory on the TMS320F28xx DSP* (SPRA958)
2. *TMS320C28x Assembly Language Tools User's Guide* (SPRU513)
3. *TMS320C28x Optimizing C/C++ Compiler User's Guide* (SPRU514)
4. *C281x C/C++ Header Files and Peripheral Examples* (SPRC097)
5. *C280x, C2801x C/C++ Header Files and Peripheral Examples Software Tools* (SPRC191)
6. *C2833x/C2823x C/C++ Header Files and Peripheral Examples Application Software* (SPRC530)
7. *TMS320F2810, TMS320F2811, TMS320F2812, TMS320C2810, TMS320C2811, TMS320C2812 Digital Signal Processors Data Manual* (SPRS174O)
8. *TMS320F2809, F2808, F2806, F2802, F2801, C2802, C2801, and F2801x DSPs Data Manual* (SPRS230)
9. *TMS320F28335, TMS320F28334, TMS320F28332 TMS320F28235, TMS320F28234, TMS320F28232 Digital Signal Controllers (DSCs) Data Manual* (SPRS439)
10. *eZdsp™ F2808 USB Technical Reference* (www.spectrumdigital.com)
11. F280x Motor-Specific Software Solutions - APSF280X (http://focus.ti.com/docs/toolsw/folders/print/apsf280x.html)
12. TMS320F281x Motor-Specific Software Solutions (http://focus.ti.com/dsp/docs/dspplatformscontento.tsp?sectionId=2&familyId=1406&tabId=2027)
13. TMS320C2000™ Controller Digital Power Software Library (http://focus.ti.com/dsp/docs/dspcontent.tsp?contentId=25262)

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Original (March 2008) to A Revision	Page
• Updated the associated zip file.	1

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265